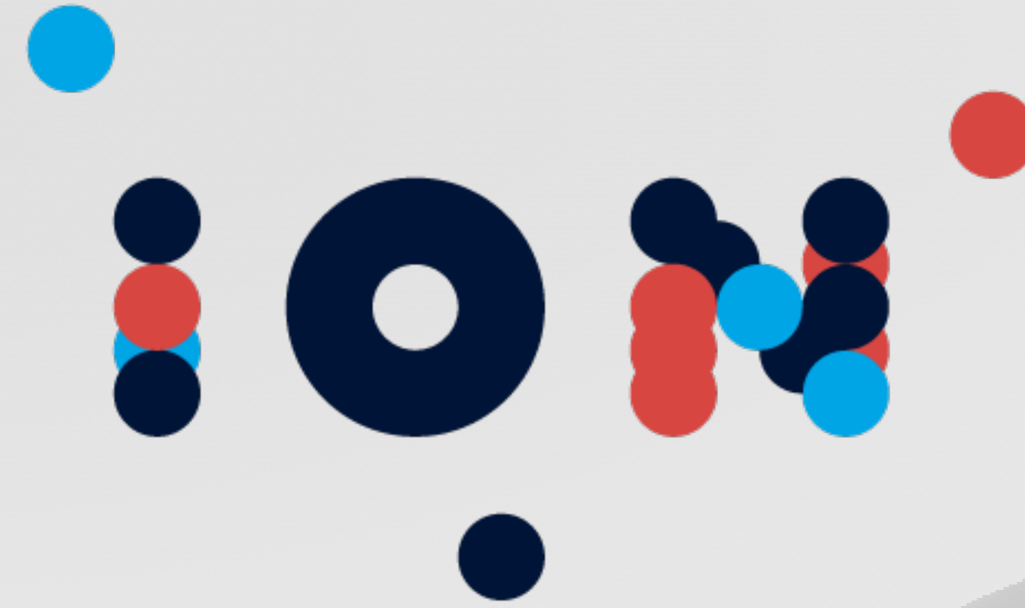
Profiling and Optimising your Go applications

Marco Marinò

About myself



- Software Engineer @ ION
- Master's degree student in AI @ University of Pisa, Italy
- Kubernetes and cloud native enthusiast
- Best paper award winner @ CLOSER 2023 Cloud Conference
  - "Semi-Automated Smell Resolution in Kubernetes-Deployed Microservices"

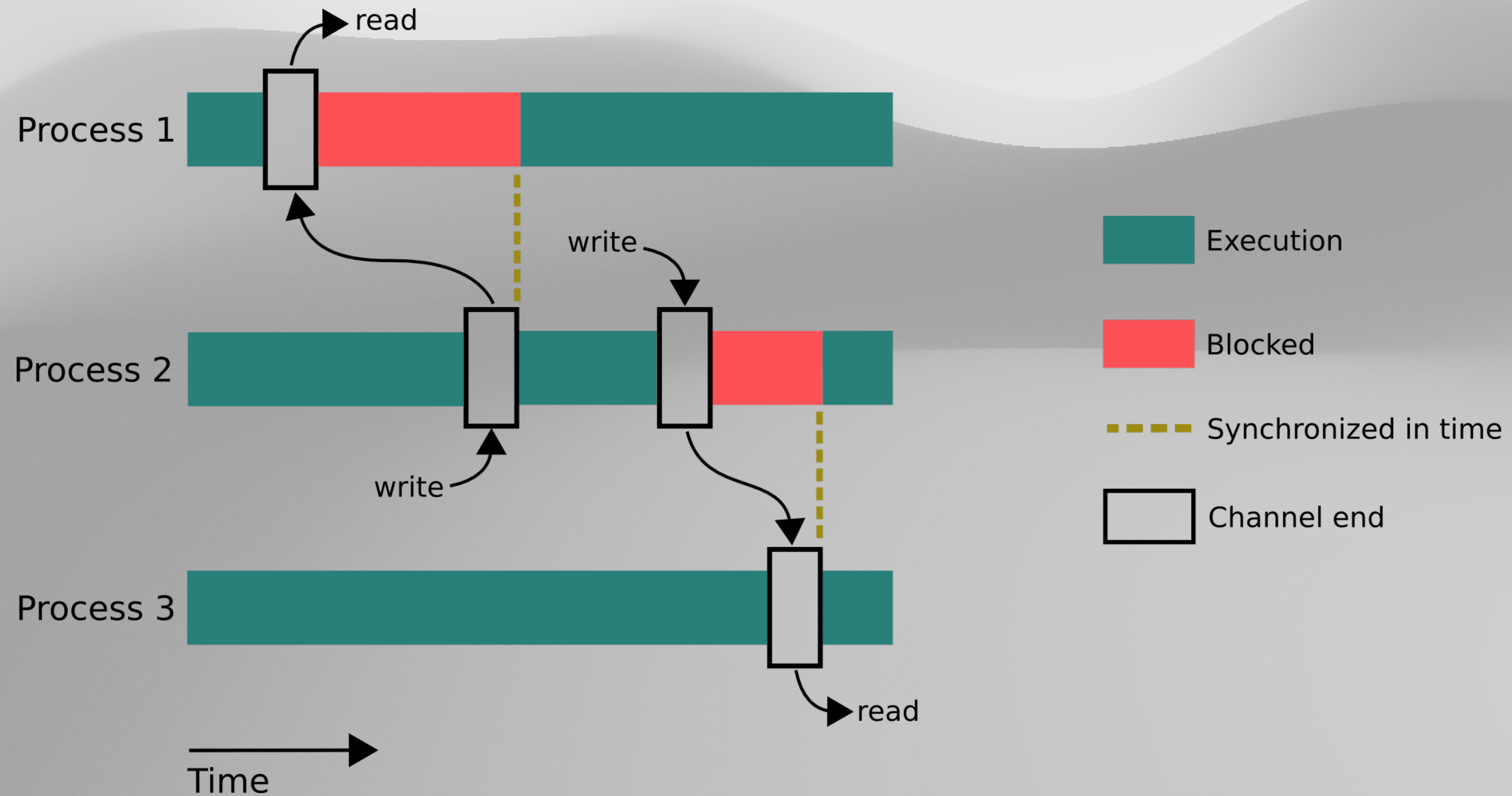
Agenda

- How Go’s Runtime Scheduler and Memory Model impact performance
- How to measure performance with benchmarks
- Leveraging pprof for in-depth profiling
- Best practices

First things first…

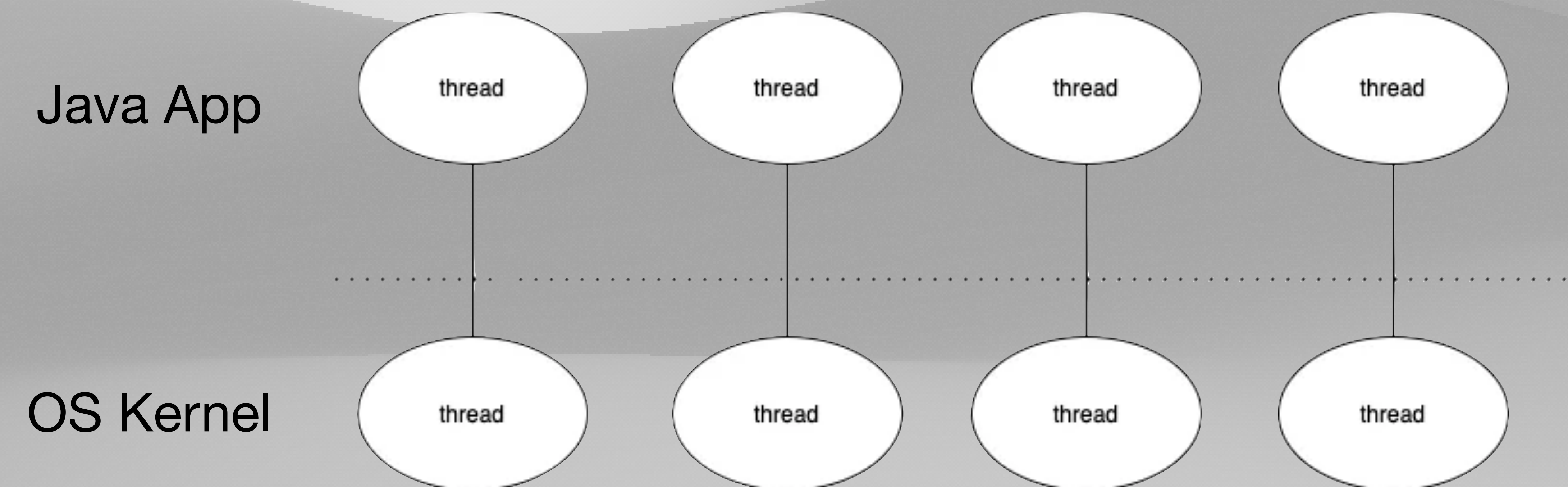
Go is a fast language… But why?

CSP (Communicating Seq. Processes)



Why a new runtime scheduler is needed?

Let's switch from Go to Java…

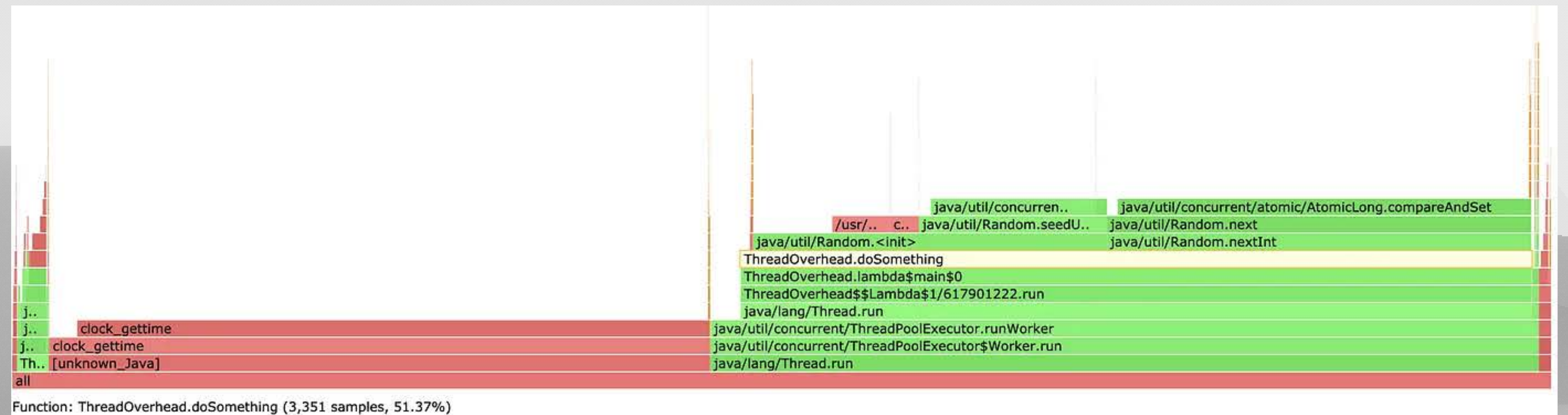


A Java Example

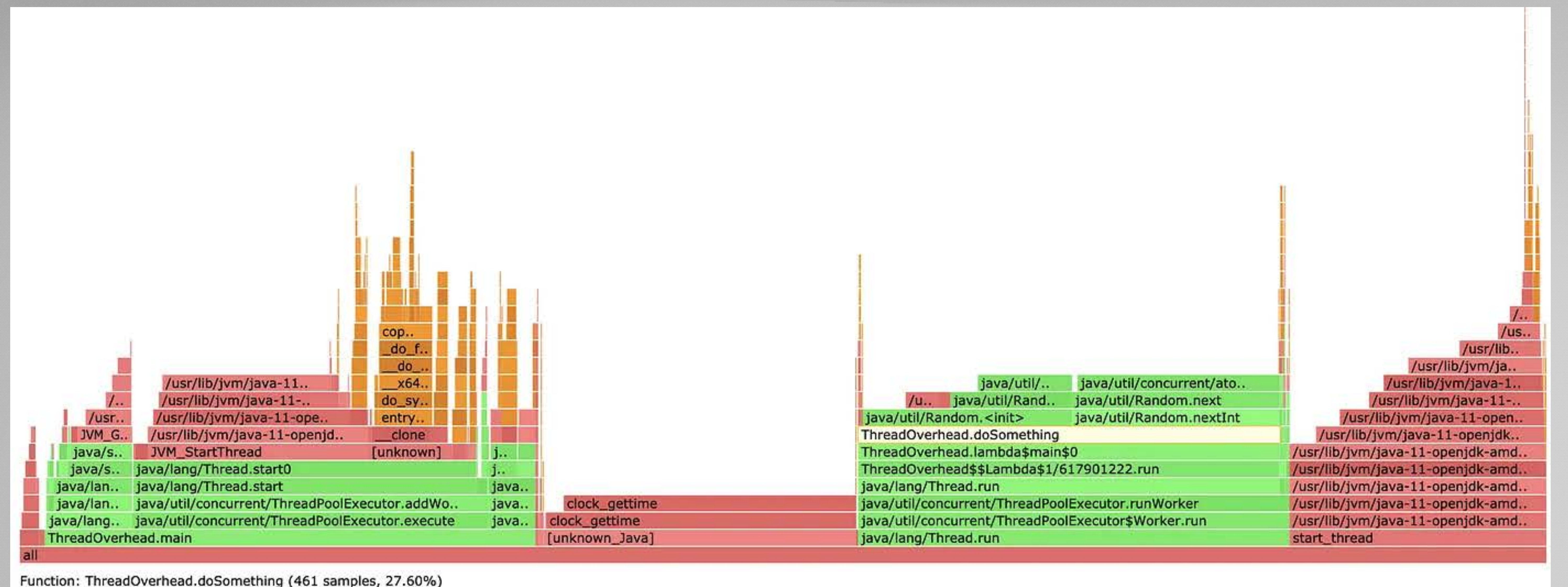
```
public static void doSomething(){
    for (int j = 0; j < 1000; j++) {
        Random random = new Random();
        int anInt = random.nextInt();
    }
}

public static void main(String []args) throws InterruptedException {
    int threadNum = Integer.parseInt(args[0]);

    ExecutorService executorService =
Executors.newFixedThreadPool(threadNum);
    for (int j = 0; j < 200000; j++) {
        executorService.execute(new Thread(() -> {
            doSomething();
        }));
    }
    executorService.shutdown();
    executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
}
```

The Challenge

#threads = 100

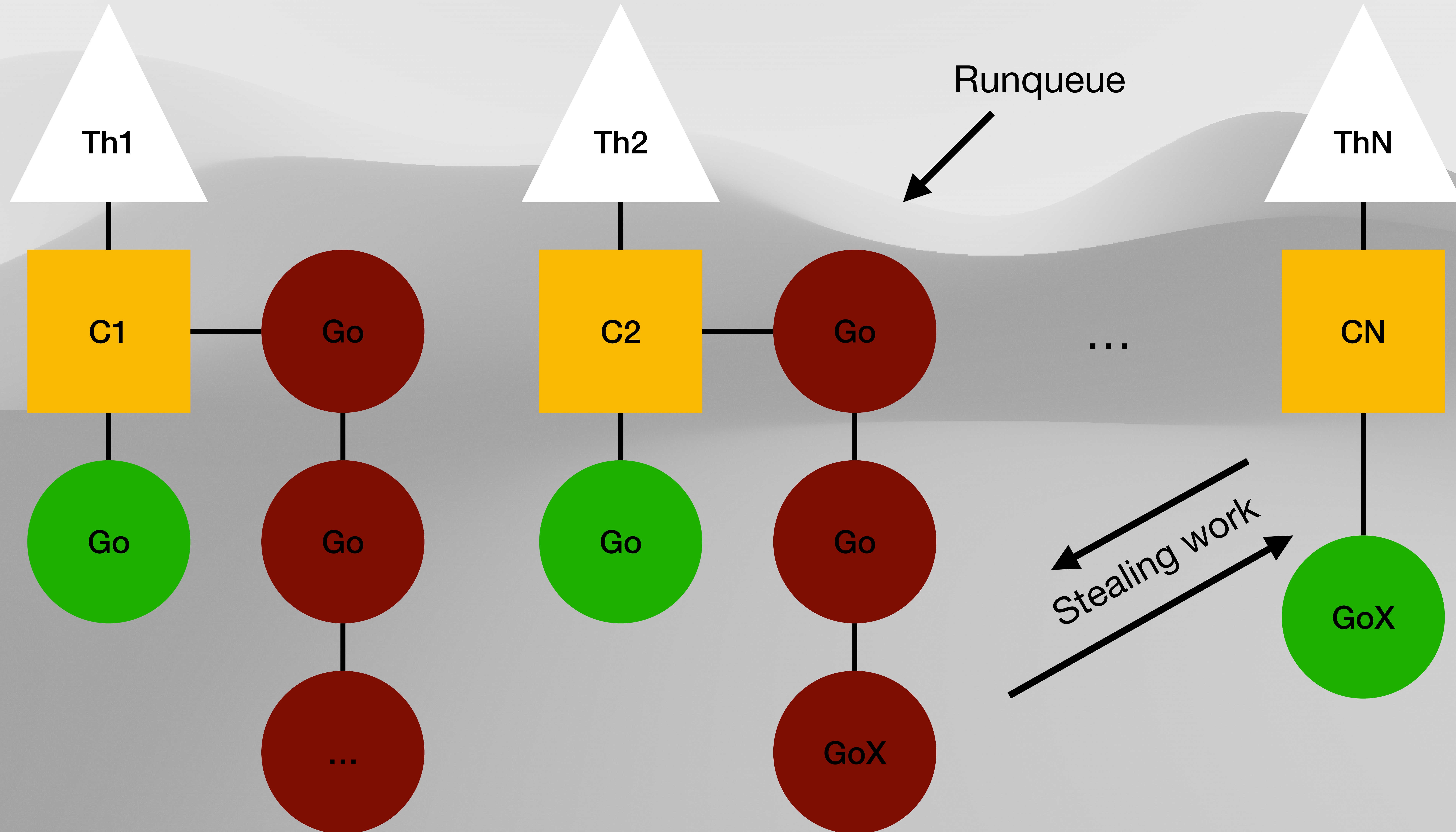


#threads = 9900



Go Runtime Scheduler

Where N = GOMAXPROCS



Another Java comparison

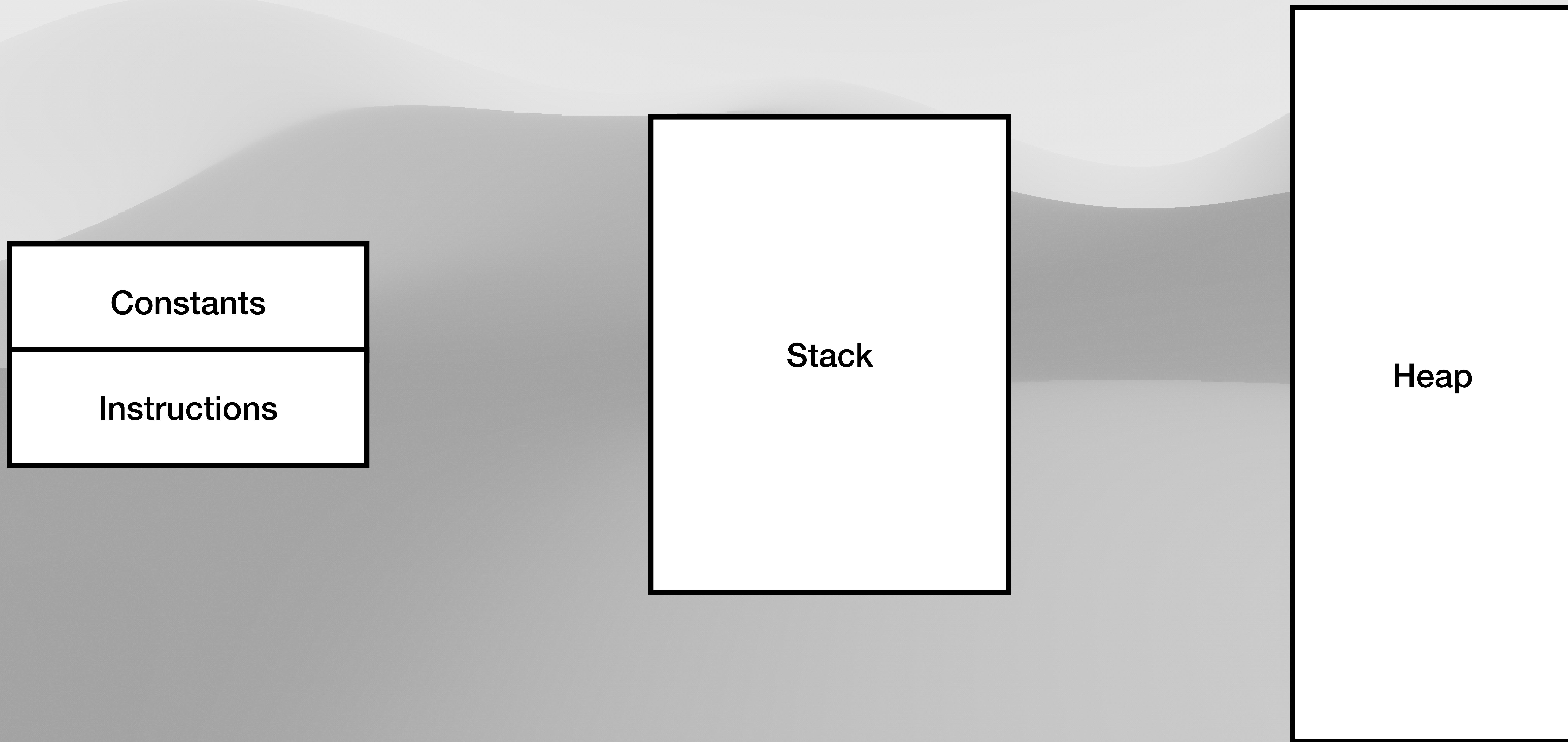
```java
public static void main(String []args) throws
InterruptedException {
    for (int i = 0 ;i<1000;i++){
        new Thread(()->{
            try {
                Thread.sleep(600000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }

    Thread.sleep(600000);
}
```

```go
func doSomething() {
    time.Sleep(10 * time.Minute)
}

func main() {
    process_id := os.Getpid()
    fmt.Println(process_id)
    for i := 0; i < 1000; i++ {
        go doSomething()
    }

    time.Sleep(10 * time.Minute)
}
```

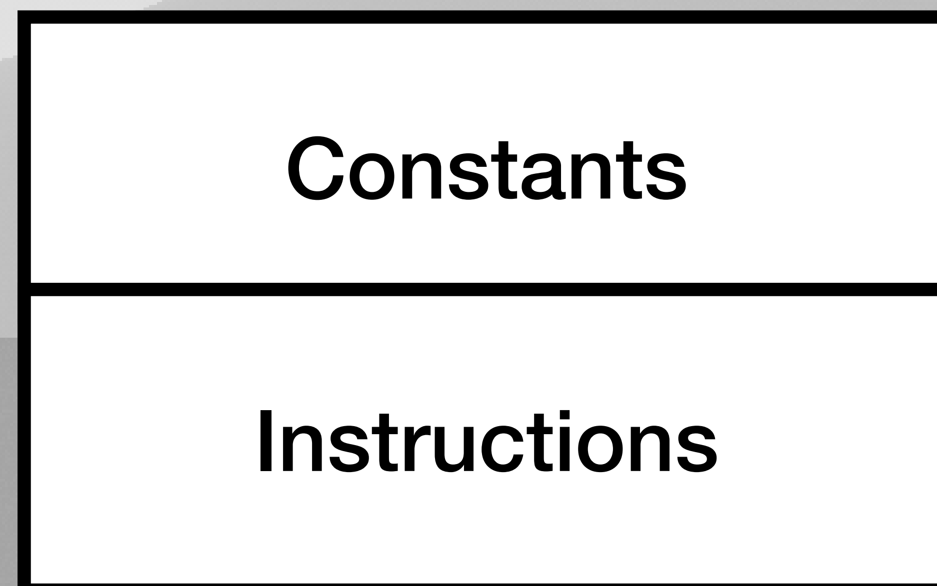
```
java-sample git:(main) ✗ ps -T 31245 | wc -l
       1018
```

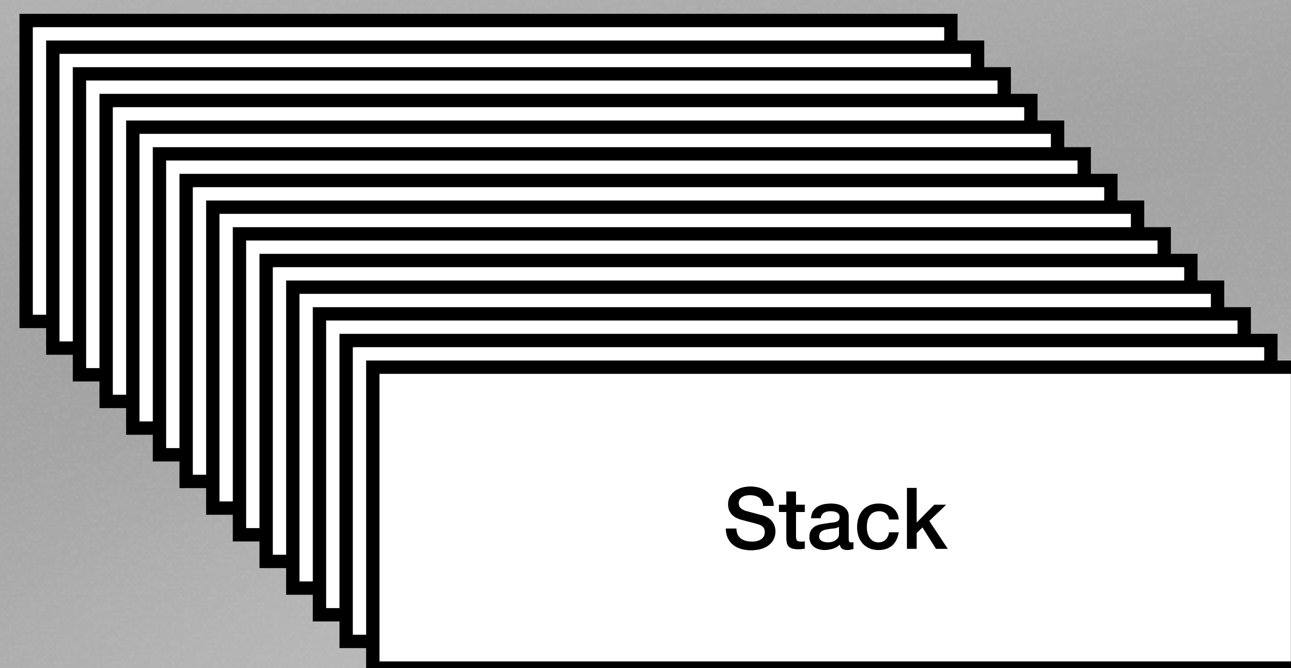
```
➜  go-speech git:(main) ✗ ps -T 32302 | wc -l
       2
```

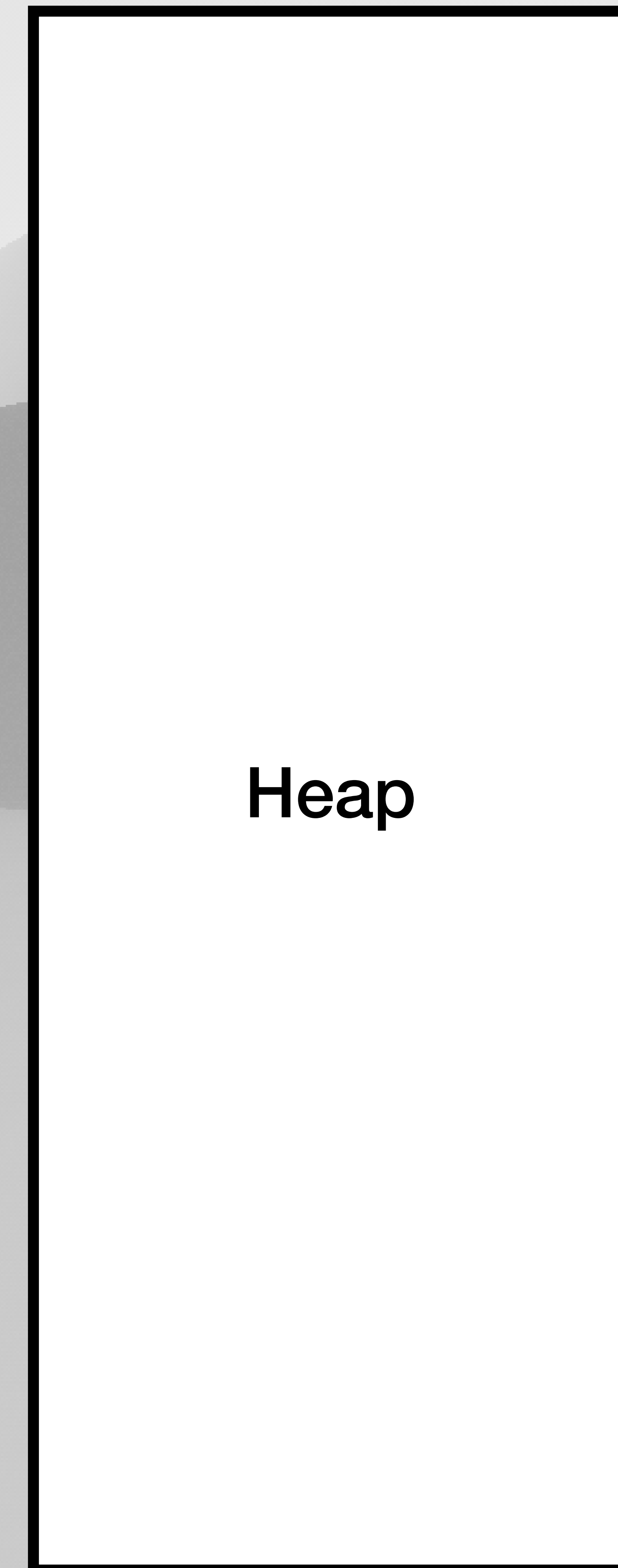
OS threads have a fixed-size stack for saving the state…



Go Memory Model







Benchmarking preconditions

- Compare two or more implementations with the most consistent environment
- Minimise the environmental impact
- It’s crucial to isolate the code being benchmarked from the rest of the program

How to write a Benchmark

- Create a file with suffix “_test.go” where to put all benchmark functions
- Each benchmark function is expected to have **func BenchmarkXxx(*testing.B)** as a signature, where *testing.B* type manages the benchmark’s timing
- b.N specifies the number of iterations, dynamically specified at runtime

Benchmarking two functions

```go
func RunPipeline1(ctx context.Context, source []string) <-chan string
{
    outputChannel := producer1(ctx, source)

    stage1Channels := []<-chan string{}

    for i := 0; i < runtime.NumCPU(); i++ {
        lowerCaseChannel := transformToLower1(ctx, outputChannel)

        stage1Channels = append(stage1Channels, lowerCaseChannel)
    }

    stage1Merged := mergeStringChans1(ctx, stage1Channels...)
    stage2Channels := []<-chan string{}

    for i := 0; i < runtime.NumCPU(); i++ {
        titleCaseChannel := transformToTitle1(ctx, stage1Merged)

        stage2Channels = append(stage2Channels, titleCaseChannel)
    }

    return mergeStringChans1(ctx, stage2Channels...)
}
```

```go
func RunPipeline2(ctx context.Context, source []string) <-chan string
{
    outputChannel := producer2(ctx, source)

    stage1Channels := []<-chan string{}

    for i := 0; i < runtime.NumCPU(); i++ {
        lowerCaseChannel := transformToLower2(ctx, outputChannel)

        stage1Channels = append(stage1Channels, lowerCaseChannel)
    }

    stage1Merged := mergeStringChans2(ctx, stage1Channels...)
    stage2Channels := []<-chan string{}

    for i := 0; i < runtime.NumCPU(); i++ {
        titleCaseChannel := transformToTitle2(ctx, stage1Merged)

        stage2Channels = append(stage2Channels, titleCaseChannel)
    }

    return mergeStringChans2(ctx, stage2Channels...)
}
```

Create and Run the benchmark functions

```go
func BenchmarkPipeline1(b *testing.B) {

    var source = generateStringSlice(30, 10)

    for i := 0; i < b.N; i++ {
        RunPipeline1(context.Background(), source)
    }

}
```

After (1) replace RunPipeline1 with RunPipeline2 in the same bench function, and run (2)

(1)

```
go test \
    -bench=BenchmarkPipeline1
\   -run=x \
    -benchmem \
    > after.bench
```

(2)

```
go test \
    -bench=BenchmarkPipeline1
\   -run=x \
    -benchmem \
    > before.bench
```

How to read a Benchmark

```
goos: darwin
goarch: arm64
pkg: my-project
BenchmarkPipeline1-10  103107 110207 ns/op 22765 B/op 1281 allocs/op
```

#Iterations Nanosec/op #bytes/op #allocs/op

Using Benchstat to compare the results

```
➜  go-speech git:(main) ✗ benchstat before.bench after.bench
goos: darwin
goarch: arm64
pkg: my-project
             │ before.bench │              after.bench               │
             │    sec/op    │   sec/op     vs base                   │
Pipeline1-10   120.14µ ± 11%   71.27µ ± 32%  -40.68% (p=0.002 n=6)

             │ before.bench │              after.bench               │
             │     B/op     │    B/op      vs base                   │
Pipeline1-10   21.74Ki ± 11%   12.64Ki ± 9%  -41.87% (p=0.002 n=6)

             │ before.bench │            after.bench              │
             │  allocs/op   │ allocs/op   vs base                 │
Pipeline1-10   1112.5 ± 15%   146.5 ± 2%  -86.83% (p=0.002 n=6)
```

Be aware of compiler optimisations

```go
var resChan <-chan string

func BenchmarkPipeline1(b *testing.B) {
	var source = generateStringSlice(30, 10)
	var rChan <-chan string

	for i := 0; i < b.N; i++ {
		rChan = RunPipeline2(context.Background(), source)
	}

	resChan = rChan

}
```

Profiling

From pprof docs…

- pprof is a tool for visualisation and analysis of profiling data
- pprof reads a collection of profiling samples in profile.proto format and generates reports
- https://developers.google.com/protocol-buffers
- Available by running: ‘go install github.com/google/pprof@latest'

Most cpu expensive tasks

```
go-speech git:(main) ✗ go tool pprof cpu1.prof
File: my-project.test
Type: cpu
Time: Apr 18, 2024 at 1:13am (CEST)
Duration: 86.68s, Total samples = 1439.83s (1661.11%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top 100
Showing nodes accounting for 1370.92s, 95.21% of 1439.83s total
Dropped 377 nodes (cum <= 7.20s)
Showing top 100 nodes out of 137
      flat  flat%   sum%        cum   cum%
   266.91s 18.54% 18.54%    266.91s 18.54%  runtime.usleep
   242.99s 16.88% 35.41%    244.41s 16.97%  runtime.(*unwinder).resolveInternal
   150.28s 10.44% 45.85%    150.30s 10.44%  runtime.readgstatus (inline)
   103.71s  7.20% 53.05%    103.71s  7.20%  runtime.memmove
    88.22s  6.13% 59.18%     88.22s  6.13%  runtime.pthread_cond_wait
    80.46s  5.59% 64.77%     80.46s  5.59%  runtime/internal/atomic.
(*Uint32).CompareAndSwap (inline)
    70.61s  4.90% 69.67%     70.61s  4.90%  runtime.(*mspan).heapBitsSmallForAddr
    66.45s  4.62% 74.29%     66.62s  4.63%  runtime.gopark
    38.78s  2.69% 76.98%    149.95s 10.41%  runtime.send
    24.56s  1.71% 78.69%     32.69s  2.27%  runtime.stackpoolalloc
    24.06s  1.67% 80.36%     24.06s  1.67%  runtime.madvise
    17.75s  1.23% 81.59%     21.11s  1.47%  runtime.(*waitq).dequeue (inline)
    16.41s  1.14% 82.73%     16.41s  1.14%  runtime.pthread_cond_signal
    15.74s  1.09% 83.82%     15.74s  1.09%  runtime.memclrNoHeapPointers
    15.50s  1.08% 84.90%     15.50s  1.08%  runtime.gcResetMarkState.func1
    15.43s  1.07% 85.97%    219.85s 15.27%  runtime.lock2
    12.82s  0.89% 86.86%     16.61s  1.15%  runtime.stackfree
    11.85s  0.82% 87.69%    340.39s 23.64%  runtime.markroot.func1
    11.11s  0.77% 88.46%    484.31s 33.64%  runtime.markroot
    10.60s  0.74% 89.19%     24.73s  1.72%  strings.ToLower
    10.43s  0.72% 89.92%     69.42s  4.82%  runtime.newproc1
     8.73s  0.61% 90.52%      8.73s  0.61%  runtime.kevent
     8.39s  0.58% 91.11%     11.66s  0.81%  strings.ToUpper
     7.77s  0.54% 91.65%      7.98s  0.55%  runtime.(*lfstack).pop (inline)
     7.40s  0.51% 92.16%      7.40s  0.51%  runtime.(*gList).pop (inline)
     7.04s  0.49% 92.65%     72.23s  5.02%  my-project.transformToTitle1.func1
     6.98s  0.48% 93.13%     20.43s  1.42%  runtime.markrootFreeGStacks
     5.98s  0.42% 93.55%     39.88s  2.77%  runtime.stackcacherefill
     4.49s  0.31% 93.86%     46.12s  3.20%  my-project.transformToLower1.func1
     2.87s   0.2% 94.06%     55.39s  3.85%  runtime.mallocgc
     ...
```

Why there's a sleep?

Why there's no track of functions with suffix 2?

functions with 2 at the end are related to RunPipeline2 and there were faster than all the methods with suffix1

What if we scroll down…

```
4.49s  0.31% 93.86%     46.12s  3.20%  my-project.transformToLower1.func1
    2.87s   0.2% 94.06%     55.39s  3.85%  runtime.mallocgc
    1.71s  0.12% 94.18%     75.04s  5.21%  runtime.scanobject
    1.46s   0.1% 94.28%    139.28s  9.67%  runtime.sellock
    1.12s 0.078% 94.36%     13.79s  0.96%  runtime.acquireSudog
    1.12s 0.078% 94.44%      8.61s   0.6%  runtime.scanblock
    0.94s 0.065% 94.50%     10.44s  0.73%  runtime.unlock2
    0.90s 0.063% 94.56%     15.97s  1.11%  runtime.casgstatus
    0.76s 0.053% 94.62%      8.75s  0.61%  runtime.getempty
    0.63s 0.044% 94.66%     73.97s  5.14%  time.Sleep
    0.58s  0.04% 94.70%    195.84s 13.60%  runtime.selectgo
    0.56s 0.039% 94.74%      8.23s  0.57%  runtime.chanrecv
    0.49s 0.034% 94.77%    265.83s 18.46%  runtime.scanstack
    0.47s 0.033% 94.81%    671.60s 46.64%  runtime.systemstack
    0.45s 0.031% 94.84%    539.34s 37.46%  runtime.gcDrain
    0.40s 0.028% 94.87%    270.40s 18.78%  runtime.schedule
    0.37s 0.026% 94.89%    243.56s 16.92%  runtime.(*unwinder).initAt
    0.36s 0.025% 94.92%     15.86s  1.10%  runtime.forEachG
    0.34s 0.024% 94.94%     20.12s  1.40%  runtime.wakep
    0.30s 0.021% 94.96%     63.44s  4.41%  runtime.ready
    0.29s  0.02% 94.98%     14.54s  1.01%  runtime.scanframeworker
    0.28s 0.019% 95.00%    134.89s  9.37%  my-project.transformToTitle2.func1
    0.27s 0.019% 95.02%     85.34s  5.93%  runtime.runqgrab
    0.26s 0.018% 95.04%    127.68s  8.87%  my-project.transformToLower2.func1
    0.25s 0.017% 95.05%     24.46s  1.70%  runtime.gfget
    0.24s 0.017% 95.07%    101.12s  7.02%  runtime.stealWork
    0.18s 0.013% 95.08%     24.44s  1.70%  runtime.deductAssistCredit
    0.17s 0.012% 95.10%     17.02s  1.18%  runtime.(*mcentral).cacheSpan
    0.17s 0.012% 95.11%    214.52s 14.90%  runtime.park_m
    0.15s  0.01% 95.12%     21.31s  1.48%  runtime.gcDrainN
    ...
```

Here are our fast stages!

Let’s dive into the code with pprof by isolating the slowest function (RunPipeline1)

*Let’s skip the time.Sleep as it was added as an example, as it present even in the faster function

```
(pprof) list my-project.transformToLower1.func1
Total: 354.25s
ROUTINE ======================== my-project.transformToLower1.func1 in /Users/marcomarino/Documents/GitHub/go-
speech/slow.go
     3.04s     75.72s (flat, cum) 21.37% of Total
      80ms       80ms     58:   go func() {
         .          .     59:           defer close(outChannel)
         .          .     60:
         .     31.98s     61:           select {
         .          .     62:           case <-ctx.Done():
         .          .     63:                   return
         .          .     64:           case s, ok := <-values:
         .          .     65:                   if ok {
         .     12.91s     66:                           time.Sleep(time.Millisecond * 800)
         .          .     67:
         .          .     68:                           res := ""
     2.95s      2.95s     69:                           for _, char := range s {
      10ms     14.82s     70:                                   res += string(unicode.ToLower(char))
         .          .     71:
         .          .     72:                           }
         .     12.93s     73:                           outChannel <- res
         .          .     74:                   } else {
         .          .     75:                           return
         .          .     76:                   }
         .          .     77:           }
         .       50ms     78:   }()
         .          .     79:
         .          .     80:   return outChannel
         .          .     81:}
         .          .     82:
         .          .     83:func transformToTitle1(ctx context.Context, values <-chan string) <-chan string {
```

Seems we’re loosing time and memory here with this ~17s part… we can do better by calling strings.toLower(s) directly…

… now let's analyse the faster one

```
(pprof) list my-project.transformToLower2.func1
Total: 285.07s
ROUTINE ======================== my-project.transformToLower2.func1 in /Users/marcomarino/Documents/GitHub/go-
speech/fast_pipeline.go
     180ms     61.52s (flat, cum) 21.58% of Total
     120ms      120ms     62:   go func() {
         .          .     63:           defer close(outChannel)
         .          .     64:
         .     21.40s     65:           select {
         .          .     66:           case <-ctx.Done():
         .          .     67:                   return
      30ms       30ms     68:           case s, ok := <-values:
         .          .     69:                   if ok {
         .     13.38s     70:                           time.Sleep(time.Millisecond * 800)
      20ms     26.57s     71:                           outChannel <- strings.ToLower(s)
         .          .     72:                   } else {
         .          .     73:                           return
         .          .     74:                   }
         .          .     75:           }
      10ms       20ms     76:   }()
         .          .     77:
         .          .     78:   return outChannel
         .          .     79:}
         .          .     80:
         .          .     81:func transformToTitle2(ctx context.Context, values <-chan string) <-chan string {
```

Now we can see there is not that useless for loop, but using the strings.ToLower(s) still has a cost of course… But overall we saved almost ~5s

… hey we also produced a memory profile !

```
(pprof) list my-project.transformToLower1.func1
Total: 7.89GB
ROUTINE ======================== my-project.transformToLower1.func1 in /Users/marcomarino/Documents/GitHub/go-
speech/slow.go
  872.52MB   974.03MB (flat, cum) 12.06% of Total
         .          .     58:   go func() {
         .          .     59:           defer close(outChannel)
         .          .     60:
   14.50MB    14.50MB     61:           select {
         .          .     62:           case <-ctx.Done():
         .          .     63:                   return
         .          .     64:           case s, ok := <-values:
         .          .     65:                   if ok {
         .   101.51MB     66:                           time.Sleep(time.Millisecond * 800)
         .          .     67:
         .          .     68:                           res := ""
         .          .     69:                           for _, char := range s {
  858.02MB   858.02MB     70:                                   res += string(unicode.ToLower(char))
         .          .     71:
         .          .     72:                           }
         .          .     73:                           outChannel <- res
         .          .     74:                   } else {
         .          .     75:                           return
(pprof) exit
➜  go-speech git:(main) ✗ go tool pprof mem2.prof
File: my-project.test
Type: alloc_space
Time: Apr 18, 2024 at 1:58am (CEST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) list my-project.transformToLower2.func1
Total: 2.89GB
ROUTINE ======================== my-project.transformToLower2.func1 in /Users/marcomarino/Documents/GitHub/go-
speech/fast_pipeline.go
    6.50MB   233.51MB (flat, cum)  7.88% of Total
         .          .     62:   go func() {
         .          .     63:           defer close(outChannel)
         .          .     64:
    6.50MB     6.50MB     65:           select {
         .          .     66:           case <-ctx.Done():
         .          .     67:                   return
         .          .     68:           case s, ok := <-values:
         .          .     69:                   if ok {
         .   162.51MB     70:                           time.Sleep(time.Millisecond * 800)
         .    64.50MB     71:                           outChannel <- strings.ToLower(s)
         .          .     72:                   } else {
         .          .     73:                           return
         .          .     74:                   }
         .          .     75:           }
         .          .     76:   }()
(pprof)
```

Here as well we can see the benefits the second function brought to us in terms of memory and number of allocations per operation…

Best practices

- Try to design your application as a pipeline of goroutines, and exploit the capability of go for scaling your goroutines!
- Use -benchtime and -count flags for your benchmarks
- Always keep track of the memory usage as it can cause a garbage collection run and therefore potential wasted time
- Try to execute benchmarks on a stable machine without having spikes during the test

Some study references…

- https://golangbyexample.com/goroutines-golang/
- https://go.dev/ref/mem
- https://www.kelche.co/blog/go/golang-scheduling/
- https://blog.logrocket.com/benchmarking-golang-improve-function-performance/
- https://github.com/google/pprof/blob/main/README.md